CS 4640: Computer Security **Spring 2022**

Homework 2 (Individual Programming Project)

Total 100 Points

This homework will contribute to 10% of your final grades.

Due date & time: 11:59 pm on February 28, 2022.

Late Policy: You have three extra days in total for all your homeworks and projects. Any portion of a day used counts as one day; that is, you have to use integer number of late days each time. If you exhaust your three late days, any late homework or project won't be graded.

Submission: Please submit your source code and instructions on how to compile your submission in ICON as a *.zip file. For Python, please mention the version you are using. The compilation instruction should be in a separate file called "README" (without the quotation).

Programming Language: You should use "Python" for this homework.

Cryptographic Library: For cryptographic library, please consider using the **PyCryptodome** library (`https://pycryptodome.readthedocs.io/en/latest/index.html`).

Command line arguments: Note that, each of the problems require the associated program to take a command line argument. You should consider using Google to check how to take command line arguments in Python. I am providing a URL that can help: `http://www.pythonforbeginners.com/system/python-sys-argv`.

Binary files: I am providing a URL that may help you figure out how to read from and write to a binary file: `https://www.tutorialsteacher.com/python/python-read-write-file`.

Problem 1 (25 pts) RC4 Stream Cipher

In this problem, your goal is to write a program that performs RC4 encryption/decryption for which it takes five command line arguments in the following order: `key-size` `key-file` `input-file` `output-file` `operation`. The first argument `key-size` will be a natural number between 40 to 128 representing the number of bits in the key. The second argument `key-file` is an input file name which will contain a key of length `key-size`. The third argument `input-file` is an input file containing the plaintext/ciphertext message you are required to encrypt/decrypt. The fourth argument `output-file` represents the output file name where you would write the output of your program. The fifth argument `operation` represents the operation ("encrypt" or "decrypt") that your program will perform.

After processing the command-line arguments, you will encrypt/decrypt the message in the `input-file` using the key in `key-file` with RC4. The resulting ciphertext/plaintext should be stored in the output file name given by the argument `output-file`.

Remember to throw away the first 3072 bytes from the pseudorandom number generator and use the rest as the key stream. The algorithm is described in: `https://en.wikipedia.org/wiki/RC4`.

One good way to test the correctness of your approach would be the following. Suppose your program is named "`rc4.py`" which is asked to encrypt a plaintext message in `a.plaintext` file with a 128-bit key stored in `a.key` file, then you invoke it twice in the following way:

```
python rc4.py 128 a.key a.plaintext a.ciphertext encrypt
```

```
python rc4.py 128 a.key a.ciphertext b.plaintext decrypt
```

If you now compare files `a.plaintext` and `b.plaintext`, then they should be identical.

Note: *When you write to the output file, please write it in the binary format not in ASCII format.* One way to verify that you are writing in the binary format to the output file is that when you open the output file in a text editor you would see weird characters. You can assume that all file contents will be in the binary format.

Problem 2 (25 pts) AES Block Cipher – CTR Mode

In this problem, your goal is to write a program that performs AES encryption/decryption in the CTR mode for which it takes five command line arguments in the following order: `key-size key-file input-file output-file operation`. The first argument `key-size` will be a constant natural number 128 representing the number of bits in the key. The second argument `key-file` is an input file name which will contain a key of length `key-size`. The third argument `input-file` is an input file containing the plaintext/ciphertext message you are required to encrypt/decrypt. The fourth argument `output-file` represents the output file name where you would write the output of your program. The fifth argument `operation` represents the operation ("encrypt" or "decrypt") that your program will perform.

After processing the command-line arguments, you will encrypt/decrypt the message in the `input-file` using the key in `key-file` with AES-CTR. The resulting ciphertext/plaintext should be stored in the output file name given by the argument `output-file`.

You should use the library function to perform AES operations in the CTR mode. You should not write your own AES encryption/decryption in the CTR mode.

One good way to test the correctness of your approach would be the following. Suppose your program is named "`aes.py`" which is asked to encrypt a plaintext message in `a.plaintext` file with a 128-bit key stored in `a.key` file, then you invoke it twice in the following way:

```
python aes.py 128 a.key a.plaintext a.ciphertext encrypt
```

```
python aes.py 128 a.key a.ciphertext b.plaintext decrypt
```

If you now compare files `a.plaintext` and `b.plaintext`, then they should be identical.

Note: *When you write to the output file, please write it in the binary format not in ASCII format.* One way to verify that you are writing in the binary format to the output file is that when you open the output file in a text editor you would see weird characters. You can assume that all file contents will be in the binary format.

Problem 3 (25 pts) RSA Encryption – Decryption

In this problem, your goal is to write a program that performs RSA encryption/decryption for which it takes four command line arguments in the following order: `operation input m n`. The first argument `operation` represents the operation ("encrypt" or "decrypt") that your program will perform. Representation of the next three arguments depends on the operation.

For encryption, `m and n` will be two prime number between 50 to 100 and be used for RSA key generation. The argument `input` will be a natural number where $0 < input < m * n$ and will represent as the plaintext message.

For decryption, `m and n` will be the private key (d and n) [based on class lecture]. The argument `input` will represent as the ciphertext message.

** After processing the command-line arguments, you will first generate RSA key based on `m`, `n` **if the operation is encryption**. During decryption, you do not need to generate any RSA key, and you can use the private key generated during encryption.

** After encryption, your output will contain the ciphertext, public key (e, n) and private key (d, n). After decryption, you just need to output the plaintext. (see the example below)

Example for testing correctness:

```
python rsa.py encrypt 5000 53 97
```

`Output:` ciphertext = 4279, Private Key = (479, 5141), Public Key = (4127, 5141)

```
python rsa.py decrypt 4279 479 5141
```

`Output:` plainttext = 5000

Note that, *these outputs are not fixed and can vary based on the implementation.*

** *You need to implement the "Extended Euclidean Algorithm" for RSA key generation.*

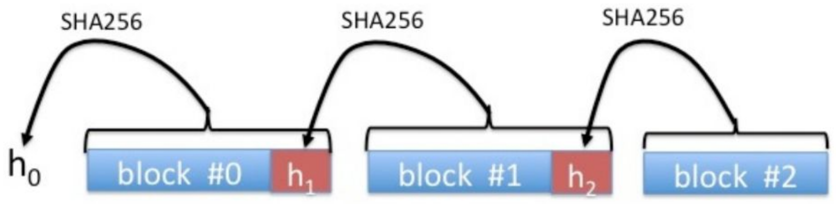
You cannot use any crypto library for this homework problem.

Details of RSA key generation, encryption, and decryption can be found in these links:

- `https://en.wikipedia.org/wiki/RSA_(cryptosystem)`
- `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`
- `https://www.di-mgt.com.au/rsa_alg.html`

Problem 4 (25 pts) Integrity Verification of a File

Suppose, a server hosts a large PDF file F that anyone can download. Clients who download the PDF need to make sure the file F is authentic before reading the content. One approach is to have the server hash the contents of F using a collision resistant hash and then distribute the resulting short hash value $h = H(F)$ to clients via some authenticated channel. Clients would download the entire file F, check that $H(F)$ is equal to the authentic hash value h and if so, read the downloaded file. However, the server in this homework adopts a different approach. Instead of computing a hash of the entire file, the server breaks the file into 4KB blocks (4096 bytes). It computes the hash of the last block and appends the value to the second to last block. It then computes the hash of this augmented second to last block and appends the resulting hash to the third block from the end. This process continues from the last block to the first. The final hash value h_0 — a hash of the first block with its appended hash — is distributed to clients via the authenticated channel as above. In this problem, you will simulate this scenario (without performing any client-server communication) to verify integrity of an input PDF file.



Your program will take two command line arguments in the following order: `input-pdf-file` and `original-hash-value`. The first argument `input-pdf-file` is an input file which represents the file F downloaded from a server. The second argument `original-hash-value` is a hex string which represents the hash value received from the server via an authenticated channel. Your program will take the `input-pdf-file`, calculate a hash of it (based on the 2nd approach mentioned above), say h_{in}, and compare whether h_{in} equals to the `original-hash-value`. If yes, your program will print `True`, otherwise will print `False`.

Example:

`python verify.py in1.pdf` fa07c3b6d8016ef612044188eacef

`Output:` False

`python verify.py in2.pdf` pq07c3b6d801213456044188eacef

`Output:` True

Note: *In this problem, you will be using SHA256 as the hash function from a standard library. When appending the hash value to each block, please append it as binary data, that is, as 32 unencoded bytes (which is 256 bits). If the file size is not a multiple of 4KB then the very last block will be shorter than 4KB, but all other blocks will be exactly 4KB.*

*** To test the correctness of your program, you are given 3 sample PDF files (e.g., input1.pdf, input2.pdf, input3.pdf) and their corresponding hash values (e.g., input1.hash, input2.hash, input3.hash) in the `inputs/` directory. Using your program, you need to figure out which PDF files are corrupted and write down your answer in the "README" file.